PROBLEM-SOLVING AND INFERENCE SOFTWARE

Ryuzo HASEGAWA and Researchers of the First Research Laboratory

Institute for New Generation Computer Technology
4-28, Mita 1-chome, Minato-ku, Tokyo 108, Japan

ABSTRACT

Problem-solving and inference software is basic software which mediates between kernel software (parallel OS) for parallel inference machines and application software. It provides a wide variety of support functions to construct application software.

The final goal of research on problem-solving and inference software is to realize cooperative problem-solving systems. With the main theme in the intermediate stage being the establishment of basic techniques for cooperative problem-solving systems, we proceeded with research and development of parallel logic programming languages, parallel programming techniques, an intelligent programming support environment, and advanced inference mechanisms and learning mechanisms.

Through this research, we developed programming techniques such as meta-programming and constraint programming which give an effective framework for cooperative problem-solving systems and a program transformation technique for the construction of efficient parallel programs.

This paper outlines the research and development of problem-solving and inference software, focusing on the work being done at ICOT.

1 INTRODUCTION

The final goal of the Fifth Generation Computer Systems (FGCS) project is to realize knowledge information processing on parallel inference machines.

In order to construct various kinds of application software for knowledge information processing using the functions provided by kernel software (parallel OS) for parallel inference machines, such as inference control and knowledge base management, we need basic software which mediates between the kernel software and application software.

Problem-solving and inference software, as well as knowledge base management software and natural language interface software, form the basic software. This software gives a wide variety of support functions necessary for the construction of application software. It also plays the role of a knowledge information processing prototype.

One basic framework required for the problem-solving and inference software is a framework for cooperative problem-solving where several agents with knowledge of different areas cooperate to solve a problem by performing their inference independently.

The framework for cooperative problem-solving is the most important base when we construct parallel application software. To establish the foundations of this framework, we need to study a model and mechanism for cooperative problem-solving and its theory. Their investigation is a research theme in problem-solving and inference software.

In order to realize this kind of framework for cooperative problem-solving, research and development of the following is required:

- A parallel computation model and parallel logic programming language which should form the basis of cooperative problem-solving;
- Programming techniques and a programming support environment for the construction of efficient programs using a parallel inference machine and the low-level function of its parallel OS;
- Advanced inference mechanisms and learning mechanisms such as induction and analogy necessary for the achievement of high-level knowledge information processing.

Taking these into consideration, we established the following three themes as the principal areas of research into problem-solving inference software in the intermediate stage:

(1) Kernel language and basic software for cooperative problem-solving

(2) Intelligent programming software

(3) Basic software for advanced inference and learning

The goal of the research on the kernel language and basic software for cooperative problem-solving is to design and develop a parallel logic programming language coordinated with problem-solving, then to establish several parallel programming techniques using it.

Through the research and development of the first version of the kernel language (KL1) [Chikayama et al. 1988], which is based on a parallel logic programming language, GHC [Ueda 1986a, Ueda 1986b], we obtained some practical knowledge of parallel inference control and implementation. At the same time, we found that some desirable functionality could not be provided in the framework of KL1, that is, the meta-function, constraint function and knowledge representation function.

Our final goal of the research on the kernel language is to develop a simple universal language with these functionalities. To achieve this goal, however, we have to delve into each function. Thus, we decided to conduct research on each of them independently for the time being, and then put them together after we have made enough progress in each one of them.

At present, we are researching how to implement the meta-function based on a reflection concept and are also researching a constraint logic programming language, CAL.

Another important area of research is to develop a program transformation technique and a partial evaluation technique, in order to implement user-level language functions and application software efficiently on the KL1 base. In the intermediate stage, we first planned to determine the foundation of these techniques based on Prolog, since the theoretical foundation, for example its program semantics, has already been established. We obtained satisfying results in this plan. We are now researching program transformation and partial evaluation systems based on GHC, and the semantics of GHC.

The research goal of intelligent programming software is to construct an intelligent programming support environment which supports the whole process all the way through, from development to maintenance of the fifth generation computer software.

Here, we aim to research software engineering based on logic programming languages. The main subject of software engineering is not coding but how to design software efficiently around coding, maintenance, and improvement.

From this point of view, we started research on a proof support system which supports mathematical proof as the research core, together with research on a prototyping support system, a Prolog programming (verification, transformation, analysis, and modification) support system, and a design visualization system giving a picture of the structure and the behavior of a program.

The main purpose of the research on the proof support system is to investigate the applicability of theorem proving techniques to programming support by researching support techniques necessary for mathematical proof. Subjects in this research are studies on a term rewriting system generator which supports inference, especially concerning equalities, and a proof compiler which generates programs from given proofs.

The research goal of the basic software for advanced inference and learning is to realize advanced inference functions used in the same way as human problem-solving and to apply them to the development of application software for knowledge information processing.

Logical deductive inference is not enough to provide computers with commonsense judgment and to make them acquire knowledge and learn in the same way as humans. We therefore need to realize inductive inference or advanced inference such as analogy. There are also many areas of human knowledge information processing which logic cannot handle. We need another approach from cognitive science which is different from logic. Therefore, we approached this theme from the angles of both logic and cognitive science.

In order to handle advanced inference in the framework of logic, we need mathematically clear formalization. From this point of view, in the approach from logic, we first tried to formalize commonsense inference using non-monotonic inference, and studied a framework to handle induction and analogy uniformly. Based on it, we are now researching the acquisition and revision of commonsense knowledge. We also researching how to make an automated generator which derives Prolog programs from examples by using a predicate generator and how to make the grammar inference algorithm efficient.

In the approach from cognitive science, we think that there are two parts of human knowledge information processing; conscious and unconscious. We are studying a cognitive model which makes inference and learning efficient based on this characterization.

The following sections describe these three themes of problem-solving and inference software in more detail, focusing on the research and development in ICOT in the intermediate stage.

2 KERNEL LANGUAGE AND BASIC SOFTWARE FOR COOPERATIVE PROBLEM-SOLVING

Research and development of the kernel language and basic software for cooperative problem-solving software is being conducted based on a parallel logic programming language, GHC. There are three goals:

(1) To investigate the provision of language functionality such as the meta-function and constraint function necessary for the construction of a high-level problem-solving system, and to clarify how to realize them, then applying these results to research the expansion of the language function of KL1 in the final stage.

(2) To establish parallel programming paradigms and parallel programming techniques, through the experimental description in GHC of typical algorithms used in various application domains.

(3) To establish program transformation techniques for constructing efficient programs and to give a formal semantics of GHC.

This section outlines research on reflection in GHC, layered-stream programming, partial evaluation, program transformation, formal semantics of GHC, and on a constraint logic programming language (CAL).

2.1 Parallel Logic Language and Reflection

Assume that we are describing an operating system for parallel computers, that is, a parallel programming system, in GHC. Such a programming system needs to input user goals, execute them as GHC processes and output the execution results. It also needs to handle meta-level concepts, such as success or failure of goal execution.

However, the current GHC does not distinguish meta-level phenomena clearly from object-level ones. This makes it difficult to describe an operating system concisely. Therefore, it is important to consider how to handle meta-level concepts which cannot fit into the current language framework of GHC. If we implement these concepts as system-defined predicates or realize them as side effects, the language may become inconsistent and the code size of the implementation may increase enormously. Therefore, we have started to study reflection as a way of handling the meta-level notion consistently.

Reflection can be considered as a function to sense the system itself and modify it dynamically. If a system or a programming language has this reflective ability, it is possible to describe a powerful operating system or problem-solving system which can flexibly perform tasks corresponding to the remaining resources and the current load.

An example of a reflective computation system is shown in Figure 1. Normally, a computation system consists of a program, data, and an executor. A computation system computes something in a certain problem domain, whereas a meta-computation system takes "another computation system" as its problem domain. A reflective system can be considered as a meta-computation system which takes itself as its problem domain and is causally connected to its data.

The concept of reflection has been proposed in FOL [Weyhrauch 1980] and 3-Lisp [Smith 1984]. In 3-Lisp, a reflective system has been realized by providing a reflective mechanism to obtain the current continuation and environment. Smith has also described the meta-circular interpreter of 3-Lisp.

The study of reflection at ICOT examines various reflective operations in GHC and, at the same time, tries to propose the new language specifications of GHC based on those arguments [Tanaka 1988]. Since we have obtained hints from Smith's approach, the basic mechanism of implementing reflection is principally the same.

Since GHC has parallelism, and meta-level phenomena are always invoked during goal execution, the implementation method becomes more complicated. Like 3-Lisp, using reflective operations to describe the GHC meta-interpreter induces the problem of a reflective tower. However, we are concentrating more on the realization of reflective operations rather than considering the problem of a reflective tower. The current status of our research is summarized below:

(1) **Stepwise enhancement of the GHC meta-interpreter**
The simplest GHC meta-interpreter can be described as a four-line program, similar to the four-line interpreter of Prolog. However, this four-line program only simulates the top-level execution of programs and cannot obtain much information from the interpreter. Therefore, we have enhanced this four-line program stepwise, and confirmed that various enhanced meta-interpreters, such as fail-safe, interruptible, scheduling and controlled meta-interpreters, are obtained from this interpreter.

- A failsafe meta-interpreter prevents the system from failing, even if a goal fails during execution;
- An interruptible meta-interpreter can suspend, resume or abort the execution of the given goal;
- A scheduling meta-interpreter enqueues the reduced goals and processes these goal sequentially

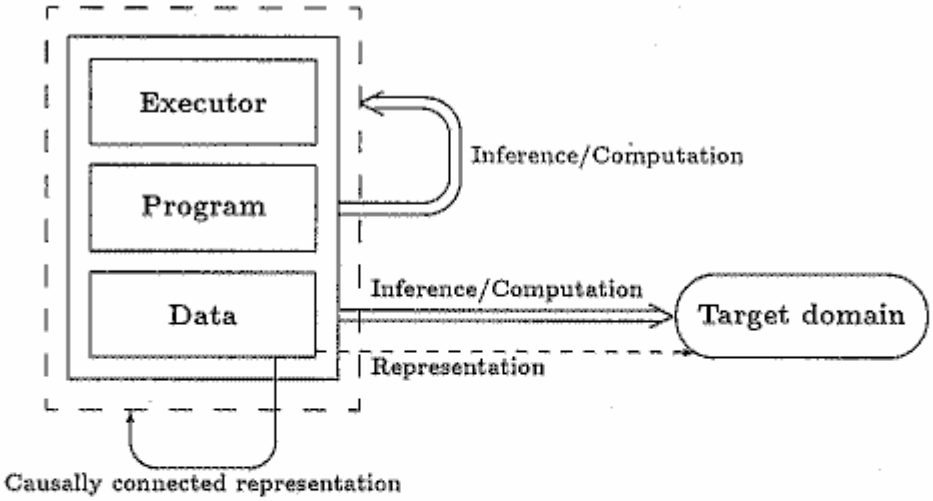

Figure 1: Reflective system

using a scheduling queue;

- A controlled meta-interpreter controls the total reduction time of a given goal using a reduction count.

(2) **Description of a variable management meta-interpreter**
To increase the expressive power of meta-interpreters further, we have developed a variable management interpreter which has the facility to manage its own local variables. In (1), we have assumed a continuation or reduction count as resources which can be controlled. Variable bindings have been added to this interpreter as resources which can be controlled by the user. We confirmed that this interpreter can run at a practical speed by running several sample programs.

(3) **Application of reflection in GHC**
We have examined the distributed computation system of GHC as an application of reflection. We assumed a system where several GHC machines are connected to each other via network managers. We are not interested in simulating the physical structures of distributed computers. Instead, our objective is to provide an abstract model of computation. We have examined the description method of a distributed computation system from the viewpoints of network managers, GHC machines and meta-interpreters. We showed that various reflective operations, such as dynamic reduction count control and load balancing, are performed on this distributed computation system.

However, the approach we have adopted so far is still very primitive. We can freely access meta-level information or resources which we would like to control. Since this seems to be very dangerous, we are currently working on the language design of Reflective GHC (RGHC) which allows more sophisticated handling of reflections.

We still have many problems as to where to position reflection in logic and how establish it in logic programming. We are planning to work on those problems and consider the semantics of reflective operations.

2.2. Constraint Logic Programming Language CAL

Constraint programming is one of the most important programming paradigms and gives a promising framework for cooperative problem solving as well as for the concept of reflection we discussed in the previous section. The most outstanding feature of constraint programming is that it allows the declarative description of problems. That is, a problem is solved by indicating a goal without reference to the method by which it should be established.

ICOT has been researching and developing a constraint logic programming language, CAL, as an element of basic software research. This subsection outlines the research and development of constraint programming languages, focusing on CAL.

Constraint logic programming languages, proposed by Colmerauer [Colmerauer 1987], and Jaffar and Lassez [Jaffar and Lassez 1987], incorporate the problem solving paradigm of constraint programming into the logic programming paradigm.

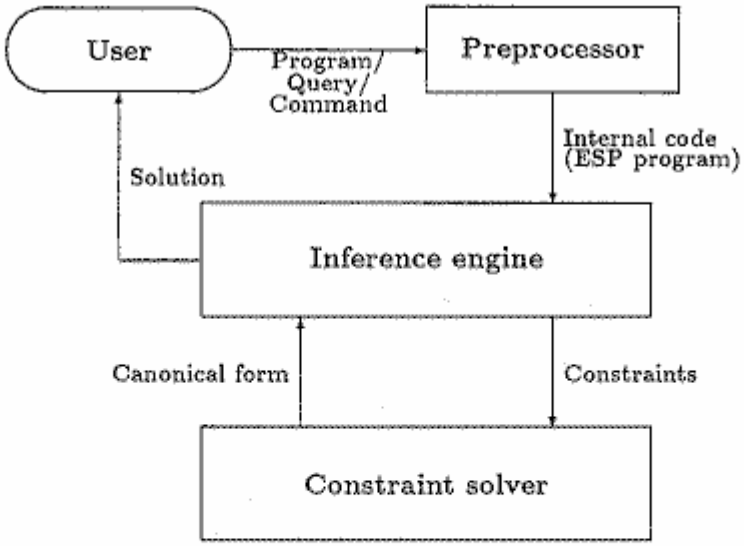

Figure 2: Organization of the CAL system

CAL aims at increasing the descriptive power of logic programming languages by replacing unification with a more powerful computation mechanism: constraints solving. Problems are described in the form of constraints, that is, relations on objects, not only of the Herbrand universe but also of other fields, and are solved by a built-in mechanism. For example, if a system of equations on real numbers occurs in a program, it is solved automatically.

CAL is a constraint logic programming language which allows users to write several types of constraints. The first prototype was implemented in 1987 on DEC2060, and, now, three systems are available on PSI, the Personal Sequential Inference machine: "Algebraic CAL" which handles linear and non-linear algebraic equations, "Boolean CAL" which handles Boolean equations, and "Typed CAL" that handles several types of constraints, including algebraic and Boolean equations, at the same time.

In Algebraic CAL, the Buchberger algorithm for computing Gröbner bases of polynomials, which has been used in recent years in computer algebra and geometry theorem proving, is utilized as the constraint solving algorithm. CAL is the first language to adopt the Buchberger algorithm as its constraint solver. This enables the system to handle non-linear equations and wield its power over a lot of algebraic problems, such as the programming for geometry theorem provers or the computation of conditional extrema by the Lagrange multiplier method.

In Boolean CAL, we use the Gröbner-base approach again. Boolean Gröbner bases can be computed by slightly modifying the Buchberger algorithm.

In Typed CAL, users can use constraints on several types of objects simultaneously. Typing is introduced to indicate the type of constraint. In the execution of a program, a suitable solver is selected automatically according to the type of each constraint.

Each of the PSI's CAL described above consists of a "preprocessor", an "inference engine," and a "constraint solver" as shown in Figure 2. The preprocessor transforms CAL programs and CAL goals (queries) to ESP programs and ESP goals. The inference engine executes ESP programs obtained by the transformation. When a constraint is detected during execution, the constraint solver is invoked to solve it. More precisely, the constraint solver collects constraints passed by the inference engine and computes the canonical form of the set of constraints.

For the final stage of the FGCS project, the geometry theorem prover has been selected as a typical application of constraint logic programming. A constraint solver that handles equations and inequations over real numbers will be investigated through this application. The hierarchical use of constraint solvers will also be investigated concurrently. A preliminary study has begun on research on parallel constraint logic programming. We intend to design a parallel constraint logic programming language with a powerful constraint solver, which will be called PCAL, based on the result of these studies.

2.3 Parallel Programming with Layered Streams

We have been studying how to write search programs in committed-choice languages (CCLs). Prolog, a sequential logic programming language, embodies unification and backtracking as its basic mechanisms, and is suitable for search problems.

Since CCLs do not have a backtracking mechanism, it is not easy to write search programs in CCLs. Solutions may be obtained by replacing some part of other solutions through backtracking. In a CCL, a process should be forked for every candidate instead of backtracking. However, structure copying is necessary for each parallel environment, which is not efficient. We have therefore proposed a data structure, called a layered stream, and a programming style based on them, called layered stream programming, for parallel processing of search problems in CCLs [Okumura and Matsumoto 1987].

The basic idea behind layered streams is to improve communication between processes by sharing some of the data structures and to achieve high parallelism. It is a generalization of the idea which is employed in the PAX parsing system [Matsumoto 1987]. In other words, PAX is a derivative variant of the method.

We have studied a way of programming search problems directly in a CCL. However, there is another way of obtaining search programs by a transformation from some specification of problems. An appropriate description language for search problems would help us to obtain such programs. We have analyzed the property of search problems and aim to devise a compiler which generates efficient codes as directly programmed.

2.4 Partial Evaluation System

The purpose of a partial evaluation system is to derive a more efficient special purpose program from a given general purpose program and its partial input, by partially performing computation on as many parts as possible using the partial input.

One of the most important applications of partial evaluation is its use in compilation, which is well known as the theory of Futamura's projection [Futamura 1971]. There has been a great deal of research on partial evaluation for this application within conventional imperative and functional languages. In particular, within Lisp, results have been obtained in compiling, compiler generation, and compiler-compiler generation by partial evaluation [Jones et al. 1985].

In logic languages such as Prolog, however, partial evaluation has recently attracted many researchers by its use in optimizing meta-programming [Levi 1986, Safra and Shapiro 1986, Takeuchi and Furukawa 1986]. Results have been reported concerning only compiling meta-programs. However, compiler generation and compiler-compiler generation remained as open problems.

The main problems to be solved for partial evaluation system are:

- Automation of the partial evaluation process;
- Making partial evaluation programs self-applicable.

By automating the partial evaluation system, we aim at making the partial evaluation process performable with less human assistance. By making the partial evaluation algorithm self-applicable, we aim at realizing compiler generation and compiler-compiler generation.

Although the implemented system has not yet succeeded in solving the automation problem, it has succeeded in making it self-applicable. Using the system, we have achieved results in compilation, compiler generation, and compiler-compiler generation. We have also succeeded in using it for incremental compilation [Fujita and Furukawa 1988].

The keys to this success are:

- Easy and sufficient use of the given partial input;
- Compactness of the partial evaluation program.

In Prolog, unification makes it very easy to utilize partial information. More concretely, due to the bi-directional nature of unification, information retained in variable bindings can be propagated both top-down and bottom-up. Secondly, it is easy to write meta-interpreters for Prolog concisely (only three lines in its simplest form). Since a partial evaluation program itself is a kind of meta-interpreter, this compactness is a great advantage in realizing self-applicability.

We shall conduct further research for the following purposes:

- Automating the above system;
- Enhancing the partial evaluation ability;
- Constructing a partial evaluation system for parallel logic languages.

In order to automate partial evaluation process, the most important problem to be solved is how to detect termination conditions for recursive user predicates. Since

the problem is undecidable in general, we more or less need an indication from programmers.

However, for a limited class of programs, it may be possible to derive determination conditions by performing sophisticated program analyses. Moreover, using mode and type information obtained by the program analyses, partial evaluation process may be made more effective. For these analyses, the abstract interpretation technique will provide a useful method.

As for improvement of the partial evaluation ability, Futamura has recently introduced the notion of generalized partial computation within a functional language [Futamura 1988]. Generalized partial computation extends the task of partial computation from mere propagation of constants and evaluation of constant expressions to propagation and stepwise reduction of constraints. This idea can be reformulated in logic programming languages. We have already obtained some results by implementing this idea.

Finally, in research on partial evaluation in parallel logic languages, we are confronted with the rather serious theoretical problem that there is no established semantics for parallel logic languages or program transformation rules that are proven to be correct.

However, we have defined a set of transformation rules called the UR-set which is rather restricted but sound in the sense that the rules never introduce a deadlock condition [Furukawa et al. 1988]. We have implemented a partial evaluation system based on the UR-set [Fujita et al. 1988]. Further research on semantics and transformation rules is in progress [Ueda 1988]. We expect that this research will contribute to a more practical partial evaluation system.

2.5 Transformation and Formal Semantics of GHC Programs

We have developed a program transformation scheme to improve the efficiency of GHC programs, and also investigated the semantics of GHC programs from the model theoretical point of view, giving an extension of the approach taken by [Apt and van Emden 1982, Lloyd 1984]. The following briefly describes this research.

2.5.1 Transformation of GHC Programs

Unfolding is a basic operation for partial evaluation and program transformation. The unfolding of Prolog programs is straightforward, and has no problem. However, it is not the case when synchronization among goals is considered. Thoughtless unfolding can cause a deadlock.

We have proposed a set of unfolding rules which does not introduce such a deadlock. The basic idea is to prohibit the unfolding of a clause with unification goals in its body if the unfolding changes the guard condition. There are four rules including auxiliary rules. The set of rules is called the UR-set.

The first rule of the UR-set is "Unification Execution/Elimination". The effect of a unification goal in the body or the guard of a clause is applied to some variables in the body. Thus, the variables may be further instantiated.

The second is "Unfolding at an Immediately Executable Goal". A clause is unfolded at a body goal if the set of candidate clauses to which the goal can commit is fixed statically.

The third is an auxiliary "Predicate Introduction and Folding". A new predicate is defined by introducing a clause whose body consists of the non-unification goals of the clause which we want to unfold. The original clause is folded by the new clause. This rule is for enabling application of the last rule.

The last rule is "Unfolding across Guard". A clause is replaced by a set of clauses if it has no unification goal. Each clause is made by unfolding the original clause at some body goal using some program clause.

The UR-set seems to be powerful enough for various applications. Recently, it was restated more formally and the folding rule was generalized [Ueda 1988]. To evaluate its effectiveness, we need to perform further experiments such as process fusion [Furukawa and Ueda 1985], the leveling of the meta-interpreter and its object program, and program synthesis from a naive definition.

To build an automatic partial evaluation system, we must find a valid control strategy to apply the UR-set. We are interested in implementing such a system in GHC. We believe it will take the form of cooperation of several unfolding processes.

2.5.2 Formal Semantics of GHC Programs

In languages such as GHC, the notion of processes which execute infinite computations controlled by guard-commit mechanisms communicating with other processes using input/output streams can be represented naturally.

In pure Horn logic programming languages, the result for declarative semantics based on the least fixpoint has been reported in [Apt and van Emden 1982,

Lloyd 1984].

In this approach, the denotation of a program is given as the minimum model of the set of Horn clauses, in other words, the set of unit clauses which is equivalent to the program. The set of unit clauses is characterized as the least fixpoint of the function obtained from the set of definite clauses. In this approach, we can characterize the set of solutions as the logical consequences of the program independently from the execution mechanism. This approach is one of the best ways of appreciating the clarity of logic programs.

We have investigated an extension of this approach to GHC programs, and presented a declarative semantics of a parallel programming language based on Horn logic such as Flat GHC [Murakami 1988]. The domain of input/output (I/O) histories has been introduced. Intuitively, an I/O history denotes an example of a computation path of a program which is generated when the program is executed without any failure or deadlock. The denotation of a program is defined as a set of I/O histories. The notion of truth is redefined for goal clauses and sets of guarded clauses. The semantics of a program is defined as the maximum model of the program.

We have also shown that the semantics is characterized as the greatest fixpoint of the function obtained from the program. Using the semantics, the solutions of programs which contain perpetual processes controlled by guard commit mechanisms can be characterized as the logical consequence of the programs.

The properties of programs which contain perpetual computation controlled by guard-commit mechanisms can be discussed using the semantics.

3 INTELLIGENT PROGRAMMING SOFTWARE

Research on intelligent programming software aims at high-level facilities from the software engineering point of view, which enables us to automate basic functions needed in each process of software development and maintenance, and to support all the processes in a uniform framework.

In the intermediate stage of the FGCS project, we have been researching basic technology, focusing on mathematical techniques such as the application of automated theorem proving, constructive mathematics, and term rewriting. An outline of this research is given below.

3.1 Computer-Aided Proof System CAP

Research and development of the computer-aided proof system (CAP) aims at technological elements such as program transformation, verification, and synthesis based on methods of automated mathematical reasoning and, thus, construction of a programming support system.

CAP will finally evolve to a cooperative problem solving system equipped with general mathematical reasoning facilities, for example, wide and deep mathematical knowledge, various utilities such as a proof editor, two-dimensional input/output, and symbolic computation.

Figure 3 shows the configuration of the CAP. It consists of a proof editor, proof checker, and proof compiler. For these components, we have been investigating the following facilities in the intermediate stage:

- A general-purpose structure editor based on user-defined grammar with various intelligent proof editing functions;
- An intelligent proof checker enabling users to write proofs easily;
- A proof compiler to construct programs from proofs with optimization functions, and an interpreter which executes constructed programs.

This subsection describes the status of each component.

3.1.1 Proof Editor

The proof editor is an intelligent editor to support description of proofs based on mathematical knowledge. It needs a user-friendly interface. We have developed a structure editor (SEMACS) with general-purpose functions which can be used by not only the CAP but also other intelligent modules such as the knowledge base system (Kappa), computer algebra system, and programming system.

SEMACS has the following features:

- Smooth interface with the text editor;
- General-purpose editor independent of grammar in the sense that it allows users to define grammar;
- Easy extension and customization;
- Guidance functions for users unfamiliar with formal grammar;

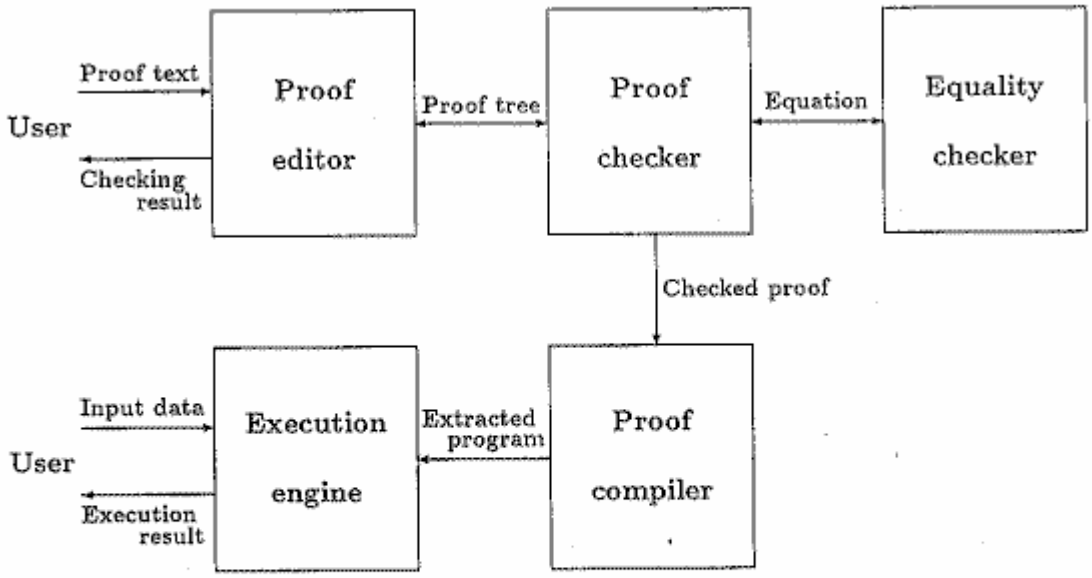

Figure 3: Configuration of the CAP system

- Natural definition and editing of list structure;
- Facility of holophrasting;
- Pretty print function which can be defined by the user;
- Editing a text containing non-terminal symbols.

Using this general-purpose structure editor, we have developed a front end for the CAP. It shows where the proof checker is currently checking in a proof text. In addition to the facility, we are now developing proof editing functions supporting interactive proof writing and checking.

3.1.2 Proof Checker

The proof checker is a kernel component of the CAP. We planned to develop a checker which can check a proof in a natural form enabling the user to write a proof easily. A prototype system, CAP-LA [Sakai 1988], has been designed and implemented according to this policy.

The current system is tuned to linear algebra for first-year university students. It checks proofs written fairly freely by users who do not know the mechanism of the proof checker, although such proofs sometimes need to be rewritten to some extent. These two features of CAP-LA — the limited target field and functions required for checking freely written proofs — accord with research and development policies. The policies are intended to develop a practical system rather than promote pure research. Other research and development policies on the system are to confirm the latest technologies such as term rewriting, automated theorem proving, logic programming, and intelligent editing, incorporating them the system.

CAP-LA checks proofs constructing the proof tree, which is based on the inference rules of natural deduction (NK), from the proofs written by the user. Generally speaking, proofs which are easily understood by the user have a lot of logical gaps. Therefore, a facility to complement them is necessary. We call this facility a proof finding facility. For first order logic, we use the Prolog theorem proving techniques. For the equations, we use the term rewriting techniques. CAP-LA has the following features:

- Separation of mathematical knowledge from the checking mechanism, providing a facility to add a checking mechanism and strategy easily;
- Environment for modifying grammar and for adding and modifying knowledge for checking;
- Ability to complete proofs through interaction with the user;
- Inference mechanism for equations using term rewriting techniques (described later);

- Automatic type checking to free the user from concern over types.

3.1.3 Proof Compiler

The proof compiler is the system which translates proofs verified by the proof checker system into programs [Takayama 1987]. This is based on the idea that a special kind of proof, called a *constructive proof* [Beeson 1985, Bishop 1967], can be seen as the description of algorithms and their verification information.

The system uses the notion of realizability interpretation [Kleene 1945, Beeson 1985, McCarty 1984], and generates executable codes from the constructive proofs of theorems. It is necessary to implement a variety of constructive logic on the proof assistant system to realize the facility. The QPC [Takayama 1988a], which is a sugared subset of QJ [Sato 1985, Sato 1986], is used as the constructive logic. QPC is the logic in which the specification, algorithms, and justification of algorithms on natural numbers and natural number lists can be described uniformly, and it is simple enough to make the research on the proof finding facilities and the extraction of efficient codes easier than other varieties of constructive logic. Tiny Quty is used as the target language of the proof compiler. The language is a subset of Quty [Sato 1987] which is also the target language of QJ.

The theoretical issues of proof compilation have been investigated, and the core part of the system has been implemented. A feasibility study has been also performed through the extraction of simple programs by the prototype system such as a gcd program for natural numbers. The following are the main research issues:

- Proof compilation algorithm based on the notion of realizability;
- Optimization of the extracted code;
- Operational semantics of Tiny Quty, that is, development of the interpreter of the language.

The first is almost completed. In the second issue, the first problem is the elimination of redundant codes. The verification information of algorithms, which is the redundant code, is extracted by the straightforward application of realizability. It causes a heavy runtime overhead, particularly on the code extracted from proofs in induction, which is generally in the form of multi-valued recursive call programs. The extended projection method (EPM), is a technique developed to eliminate the redundant code [Takayama 1988b]. The idea of the EPM is to analyze and eliminate the redundancy at each step of the proof which makes the procedure easier and more effective than the traditional syntactic optimization technique. For higher level optimization of algorithms, proof normalization, which is a well-known notion in the field of proof theory [Prawitz 1965], proved to be effective to some extent. In the last issue, the interpreter was implemented experimentally [Takayama 1987, Takayama 1988a, Takayama 1988b].

The next stage of research will deal with a more general-purpose proof assistant environment. The following themes have been set for this goal:

- Development of an interactive proof assistant environment, and the improvement of the proof editor;
- Introducing higher order features to the proof description language to describe a larger area of mathematics than linear algebra, and enhancing the proof checking facilities;
- Development of the mathematics knowledge base for the improvement of proof assistant facilities.

The main research themes in the final stage of the FGCS project will be as follows: the first is the improvement of the proof assistant system to make it practical. This research which will be along the same lines as current research. Another research theme is to develop an advanced parallel programming environment by using techniques developed for the proof assistant system.

3.2 Term Rewriting System Metis

Metis [Ohsuga and Sakai 1986] supports specific mathematical reasoning, that is, inference associated with the equal sign (=). Such inference is, in general, intricate and complicated, thus invoking an urgent need for machine support. Metis provides an experimental environment for studying practical techniques of equational reasoning. The policy of developing Metis enables implementation, testing, and evaluation of the latest techniques for inference as rapidly and freely as possible. Therefore, we decided to develop a system separate from CAP and intended to incorporate only practical techniques established on Metis in CAP whenever necessary.

We adopted the term rewriting system (TRS) as a basic technique to handle equations. The TRS is a set of oriented equations, called rewrite rules, and rewrites a term replacing the left-hand side by the right-hand side of a rewrite rule. There are main two reasons for our selection of TRS: (1) it is easy to handle by machine and can be efficient, and (2) there are quite a few studies of TRSs from the theoretical point of view, especially studies of termination and confluence property, which is important for the computation mechanism.

The kernel function of Metis is the *Knuth-Bendix* (KB) *completion procedure* [Knuth and Bendix 1970]. Roughly speaking, the KB procedure consists of two processes: (1) the orientation process of equations to assure the termination of rewriting using the semantic ordering or syntactical ordering method, and (2) the superposition process to make TRS confluent generating critical pairs (CPs) as new equations which represent ambiguity between rewrite rules. By iterating these two processes, a complete (terminating and confluent) TRS can be obtained.

However, two major problems are encountered during the KB process. One is the emergence of unorientable CPs in the superposition process. The other is the generation of infinitely many CPs. In neither case can we obtain a complete TRS. We solved the first problem by converting unorientable equations to orientation-free rewrite rules which can be applied either left to right or right to left. This extended procedure, that is, the KB procedure with orientation-free rewrite rules, is called *unfailing KB*. To solve the second problem, we adopted an extension of the KB procedure, called the *S-strategy* [Hsiang and Rusinowitch 1987]. The S-strategy determines whether a given equation is a theorem of the equational theory instead of obtaining a complete TRS and is complete in the sense of refutational theorem proving.

Research and development of Metis on how equational inference can become more efficient without loss of completeness is a long-range project. We are considering this from several points of view: implementation techniques [Ohsuga and Sakai 1988], theoretical view point, and user interface. We are planning to associate Metis with a knowledge base such as Kappa to handle the enormous number of rewrite rules which may be required in the future.

4 BASIC SOFTWARE FOR ADVANCED INFERENCE AND LEARNING

The aims of the study on the basic software for advanced inference and learning are to provide an advanced inference mechanism such as commonsense reasoning, which cannot be achieved by ordinary deductive inference, and knowledge acquisition and learning mechanisms which are essential for building large knowledge information systems. In the intermediate stage, we have been conducting basic research to achieve the above goal and have taken two approaches: logical and cognitive.

In the logical approach, we have been investigating three themes: (1) general formalization of commonsense reasoning, (2) a method for the revision and acquisition of commonsense knowledge, and (3) inductive inference based on the model theory.

In (1), we have developed a unified framework for advanced inference methods such as induction and analogy. In (2), we have been doing research focused on default reasoning, which is a subclass of commonsense reasoning, on formalized revision, and on acquisition of commonsense knowledge. In (3), we have investigated the problem of how to generate new predicates.

In the cognitive approach, we have constructed a cognitive model of conscious/unconscious processing, and simulated the model in a parallel logic programming language. The model consists of two closely interactive parts: symbol processing and pattern processing. One of the parallel symbol processes is executed consciously (conscious processing), and all the other processes are executed automatically (unconscious processing). The following subsections briefly describe these studies.

4.1 General Formalization of Commonsense Reasoning

We believe that human commonsense reasoning is supported by advanced inference mechanisms such as induction, analogy, and default reasoning. We have been studying formalization of commonsense reasoning for mathematical discussion and have developed a unified framework for various advanced inferences.

The unified framework is possible by regarding advanced inference as nonmonotonic reasoning. One of the formalizations of the advanced inference is circumscription by J. McCarthy [McCarthy 1980, McCarthy 1986]. Circumscription formalizes the notion of closed world assumption, that is, "A property is satisfied by only those entities which are explicitly stated so ". However, circumscription does not successfully formalize those inferences which generalize knowledge, such as induction and analogy.

Therefore, we have formalized the following notion: "When all the demonstrated instances of predicate P are positive instances of ψ, we can assume that all instances of P satisfy ψ (When all the instances that proved to have a property P have a property Ψ, all instances having a property P have a property ψ [Arima 1988b].)" This formalization is called *ascription* and is a formalization of induction and analogy.

Advanced inference can be formalized as inferring on the *most preferred models* by introducing a *preference* order over models. Unlike circumscription, ascription has a discrete preference order, and performs radical belief revision. Therefore, ascription is also suitable for representing management mechanisms for hypotheses which are produced by the intelligent system itself.

4.2 Acquisition and Revision of Commonsense Knowledge

In realizing ascription which is a unified framework for commonsense reasoning, how to provide a concrete preference order is a problem. We have studied the human preference order in default reasoning. Default reasoning infers the most plausible result from the commonsense knowledge which is regarded as usually correct knowledge even though there are a few exceptions. The following subsection looks at acquisition and revision methods for default reasoning.

4.2.1 Acquisition of Commonsense Knowledge

Since commonsense varies with historical, geographical, social and individual background, intelligent systems need the ability to acquire commonsense corresponding to different contexts. For example, if they can acquire individual commonsense that users have, they provide user-friendly environments which interpret the users' intention appropriately. From this point of view, we have taken the first step forward away from current research, which assumes that commonsense is provided in advance, towards the future research on acquiring commonsense knowledge.

The idea of the research [Arima 1988a] is intuitively explained as follows: "If entities in a class which are shown to have a property are much more numerous than entities which are shown not to have that property, we can acquire commonsense that the property is usually satisfied in the class."

We have two theoretical problems to perform such commonsense acquisition. They are:

(1) Representation of commonsense knowledge varying with classes;

(2) Representation of an overwhelming majority.

For (1), we have proposed *partially directional circumscription*, a specialized version of *formula circumscription* [McCarthy 1986] which is a general form of circumscription. For (2), we have introduced the *surpassing relation*, a binary relation over predicates.

We now plan to clarify problems for this approach and investigate application and cooperation with ascription.

4.2.2 Revision of Commonsense Knowledge

The idea of the revision method of commonsense knowledge is related to the study on the famous example of default reasoning called the *Yale Shooting Problem* [Hanks and McDermott 1986].

Hanks and McDermott evaluated the current formalization of the default reasoning on the temporal projection and showed that no current formalization captures human commonsense.

We have taken an approach based on *minimal change* for the Yale Shooting Problem. The formalization of minimal change states that humans infer by commonsense that a set of facts in a new situation is changed minimally from the set of facts in the previous one to preserve consistency. We have given approximate solutions for the Yale Shooting Problem and a similar problem in the inheritance system [Satoh 1987].

We have applied this formalization to the revision method of commonsense knowledge. We have developed a formalization of revision strategy which performs minimal revision, that is, to treat contradictory knowledge as exceptions when it is added to current commonsense knowledge [Satoh 1988].

4.3 Inductive Inference Based on the Model Theory

Shapiro's model inference [Shapiro 1982] gives a very important strategy for inductive inference based on the model theory for logic programs. In the model inference, however, there are very strong assumptions, as follows. Finitely many predicates, which are sufficient for describing a target program, are given in advance. Furthermore, it is assumed that an oracle which gives input/output examples of the program knows the intended interpretation of all the predicates. This means that the ability of the inference system very much depends on the user's programming knowledge.

Recently, several approaches to the problem have been made, in which an inference system generates new predicates by itself [Muggleton and Buntine 1988]. In such approaches, it is important to handle the following problems:

(1) When will be a predicate generated?

(2) What is the meaning of the new predicate?

To deal with these problems, we consider a class of logic programs, which are sufficiently and syntactically restricted, as a target of inference. Ishizaka [Ishizaka 1988] gives an efficient algorithm for inferring one such class, DRLP, which is equivalent to the class of finite state acceptor. We will try to extend this class to deal with more general logic programs.

4.4 Cognitive Model of Conscious and Unconscious Processing

We understand that the basic problems in realizing artificial intelligence are knowledge acquisition (or learning) and efficient extraction of acquired knowledge. Realizing their importance, we proposed a cognitive model of conscious/unconscious processing (C/U model) [Oka 1987, Oka 1988].

The model consists of two closely interactive parts: symbol processing and pattern processing. In symbol processing, at most one of the parallel processes is executed consciously (conscious processing) and the others are executed automatically (unconscious symbol processing). Although symbol processing proceeds deterministically, pseudo-backtracking is available in conscious processing using recent memory. Pattern processing is spreading activation in a network, which is executed unconsciously (unconscious pattern processing).

We simulated the model in a parallel logic programming language GHC utilizing the characteristics of the language. That is, we noticed the correspondence between the basic characteristics of the model and that of the language: AND-parallelism, choice nondeterminism, and the suspension rule. Utilising these characteristics of the language as it is, we added the following functions:

(1) Narrowing down OR candidates with pattern processing;

(2) Enabling pseudo-backtracking with recent memory.

Pattern processing is simulated using the language as a process description language.

We started simulation from the part of interaction between conscious processing and unconscious pattern processing. As an example for simulation, we took up the process of doing a task of selecting a disparate one of a few items, for example, {run, write, pick, eat}.

The process of doing this kind of task consists of conscious processing and unconscious pattern processing. That is, firstly, a property for a classification occurs unconsciously, according to the problem, context, and the solver's explicit and implicit knowledge which reflects his experience. Secondly, the property of each item is checked consciously. If exactly one item is disparate on the property, it is the answer. If not, another property occurs and it is checked. Conscious processing is efficient because it deals only with properties that have occurred; that is, knowledge which can be accessed from conscious processing is narrowed down by unconscious processing.

In the model, tasks can be shared between symbol processing and pattern processing, making the best use of each part; moreover, inference and learning in each part are expected to become more efficient through the interaction of each part.

5 CONCLUSION

Our final goal of the research and development of problem-solving and inference software is to develop a cooperative problem-solving system which supports the construction of many kinds of application software. One of the main themes in the intermediate stage is parallelization which is essential to the development of such a system. We obtained fundamental results in this area.

Meta-programming by reflection and constraint logic programming will be important paradigms to make schemes of knowledge representing languages which should be developed using the kernel language KL1. We believe that the meta-function and constraint-function realized by these paradigms gives a common base for the cooperative problem-solving system.

Program transformation techniques and partial evaluation techniques based on a parallel logic programming language GHC are almost completely developed. In the research on the proof support system, CAP, we developed a good amount of theoretical background and implemented many tools.

Advanced inference and learning is one of the most important themes of the FGCS project. To achieve progress in these areas, however, it is necessary to make it clear what human knowledge information processing is, which is a very difficult problem. There is no world-wide approved standard method to study it yet. At present, we are investigating this theme based on its mathematical model.

Although this paper did not discuss other related research conducted by the First Research Laboratory because of limited space, much has been done. Related research includes ARGUS [Kanamori et al. 1986, Kanamori and Horiuchi 1984, Kanamori and Horiuchi 1986], a program verification, transformation, synthesis and analysis system; ANDOR [Takeuchi et al. 1987a, Takeuchi et al. 1987b], a parallel problem-solving language for concurrent systems; EUODHILOS [Sawamura and Minami 1988, Sawamura et al. 1988], a computer aided reasoning system; and MENDEL [Honiden et al. 1985, Honiden et al. 1986, Uchihira et al. 1987], a prototyping support system.

In the final stage, we will concentrate on parallelization. As part of the research on intelligent programming software, we are planning to develop (1) a parallel

knowledge programming language submodule, (2) a parallel intelligent programming support submodule, (3) a proof support submodule and (4) advanced inference and learning mechanisms.

For the parallel knowledge programming language submodule, we will conduct further research on metaprogramming, constraint logic programming and semantics based on a parallel logic programming language. Considering these results, we plan to extend and improve KL1.

For the parallel intelligent programming support submodule, we will continue basic research on program transformation and verification of parallel logic programs, then develop a practical partial evaluation system and an interactive transformation, synthesis and verification system considering flexibility and extensibility.

For the proof support submodule, we will enrich the practicality of the proof support system (CAP) developed in the intermediate stage, then expand it to a parallel algorithm design support system to develop an intelligent support environment for parallel programming.

For the advanced inference and learning mechanisms, we plan to proceed with research on the formalization of commonsense reasoning, a predicate generator based on inductive inference, and a model of cognition. Then, cooperating with research on natural language processing and expert systems in the other laboratories, we will develop them as integrated research on learning from the viewpoints of both theory and application.

ACKNOWLEDGMENTS

The research on the problem-solving and inference software was carried out by the first research laboratory at ICOT in tight cooperation with six manufactures. Thanks are firstly due to who have given support and helpful comments, including Dr. Fuchi, the director of the research laboratories at ICOT, Mr. Yokoi, the former chief of the second research laboratory at ICOT and the current director of EDR, and Dr. Furukawa, the deputy director of the research laboratories at ICOT. Many fruitful discussions were done at the meetings of Working Groups: PPS, SYC, and FAI. Special thanks go to many people at the cooperating manufacturers in charge of the joint research programs.

REFERENCES

[Apt and van Emden 1982] Apt, K. and van Emden, M. H., Contributions to the theory of logic programming, *J. ACM*, 29, 1982

[Arima 1988a] Arima, J., Generating Rules with Exceptions, in this volume, 1988

[Arima 1988b] Arima, J., Formalization of Advanced Inference Processing as Nonmonotonic Reasoning (in preparation)

[Beeson 1985] Beeson, M. J., *Foundations of constructive mathematics*, Springer-Verlag, 1985

[Bishop 1967] Bishop, E., *Foundation of constructive analysis*, McGraw-Hill, New York, 1967

[Chikayama et al. 1988] Chikayama, T. et al., Overview of the Parallel Inference Machine Operating System (PIMOS), in this volume, 1988

[Colmerauer 1987] Colmerauer, A., Introduction to Prolog-III, in *ESPRIT'87, Achievements and Impact, Proc. 4th Annual ESPRIT Conference*, pp.28–29, Brussels, North-Holland, 1987

[Fujita and Furukawa 1988] Fujita, H. and Furukawa, K., A Self-Applicable Partial Evaluator and Its Use in Incremental Compilation, *New Generation Computing*, 6(2,3), June 1988

[Fujita et al. 1988] Fujita, H. Okumura, A. and Furukawa, K., Partial Evaluation of GHC Programs Based on the UR-set with Constraints, in *Proc. Fifth International Conference and Symposium on Logic Programming*, Seattle, 1988

[Furukawa and Ueda 1985] Furukawa, K. and Ueda, K., GHC Process Fusion by Program Transformation, in *2nd Conf. Proc. Japan Soc. Softw. Sc. Tech.*, Tokyo, 1985

[Furukawa et al. 1988] Furukawa, K., Okumura, A., and Murakami, M., Unfolding Rules for GHC Programs, *New Generation Computing*, 6(2,3), June 1988

[Futamura 1971] Futamura, Y., Partial Evaluation of Computation Process – An Approach to a Compiler-Compiler, *Systems, Computers, Controls*, 2(5):45–50, 1971

[Futamura 1988] Futamura, Y., Generalized Partial Computation, in D. Bjørner, A. P. Ershov, and N. D. Jones, editors, *Partial Evaluation and Mixed Computation*, North-Holland, 1988

[Hanks and McDermott 1986] Hanks, S. and McDermott, D., Default reasoning, nonmonotonic logics, and the frame problem, in *Proc. AAAI86*, pp.328–333, 1986

[Honiden et al. 1985] Honiden, S., Uchihira, N., and Kasuya, T., Software Prototyping with *MENDEL*, in *Proc. Logic Programming 85*, LNCS-221, pp.108–116, Springer-Verlag, 1985

[Honiden et al. 1986] Honiden, S., Uchihira, N., and Kasuya, T., MENDEL: Prolog based concurrent object oriented language, in *Proc. COMPCON 86*, pp.230–234, 1986

[Hsiang and Rusinowitch 1987] Hsiang, J. and Rusinowitch, M., On Word Problems in Equational Theories, in *ICALP, 14th International Colloquium Automata, Languages and Programming*, pp.54–71, 1987

[Ishizaka 1988] Ishizaka, H., Inductive inference of regular languages based on model inference, in *Proc. Logic Programming Conference '87*, LNCS-315, pp.178–184, Springer-Verlag, 1988

[Jaffar and Lassez 1987] Jaffar, J. and Lassez, J-L., Constraint Logic Programming, in *Proc. 4th IEEE Symposium on Logic Programming*, 1987

[Jones et al. 1985] Jones, N. D., Sestoft, P., and Søndergaard, H., An Experiment in Partial Evaluation: The Generation of a Compiler Generator, in J.-P. Jouannaud, editor, *Rewriting Techniques and Applications*, LNCS-202, pp.124–140, Springer-Verlag, 1985

[Kanamori and Horiuchi 1984] Kanamori, T. and Horiuchi, K., Type Inference in Prolog and Its Applications, Tech. Report TR-095, ICOT, 1984, also in *Proc. 9th International Joint Conference on Artificial Intelligence*, pp.704–707, 1985

[Kanamori and Horiuchi 1986] Kanamori, T. and Horiuchi, K., Construction of Logic Programs Based on Generalized Unfold/Fold Rules, Tech. Report TR-177, ICOT, 1986, also in *Proc. 4th International Conference on Logic Programming*, pp.744–768, 1987

[Kanamori et al. 1986] Kanamori, T., Fujita, H., Seki, H., Horiuchi, K., and Maeji, M., Argus/V: A System for Verification of Prolog Programs, in *Proc. FJCC*, Dallas, Texas, IEEE Computer Society Press, 1986

[Kleene 1945] Kleene, S. C., On the interpretation of intuitionistic number theory, *J. of Symbolic Logic*, 10:109–124, 1945

[Knuth and Bendix 1970] Knuth, D. E. and Bendix, P. B., Simple word problems in universal algebras, in J. Leech, editor, *Computational problems in abstract algebra*, pp.263–297, Pergamon Press, Oxford, 1970, also in Siekmann and Wrightson, editors, *Automation of Reasoning 2*, pp.342-376, Springer-Verlag, 1983

[Levi 1986] Levi, G., Object Level Reflection of Inference Rules by Partial Evaluation (extended abstract), in P. Maes and D. Nardi, editors, *Workshop on Meta-Level Architectures and Reflection*, Sardinia, North-Holland, 1986

[Lloyd 1984] Lloyd, J. W., *Foundations of logic programming*, Springer-Verlag, 1984

[Matsumoto 1987] Matsumoto, Y., A Parallel Parsing System for Natural Language Analysis, *New Generation Computing*, 5(1):63–78, 1987

[McCarthy 1980] McCarthy, J., Circumscription — a form of non-monotonic reasoning, *Artif. Intell.*, 13:27–39, 1980

[McCarthy 1986] McCarthy, J., Application of Circumscription to Formalizing Common-sense Knowledge, *Artif. Intell.*, 28:89–116, 1986

[McCarty 1984] McCarty, D. C., *Realizability and Recursive Mathematics*, Ph.D thesis, Oxford, 1984

[Muggleton and Buntine 1988] Muggleton, S. and Buntine, W., Towards Constructive Induction in First-order Predicate Calculus, TIRM 88-031, The Turing Institute, 1988

[Murakami 1988] Murakami, M., A New Declarative Semantics of Parallel Logic Programs with Perpetual Processes, in this volume, 1988

[Ohsuga and Sakai 1986] Ohsuga, A. and Sakai, K., Metis: A Term Rewriting System Generator, in *Symposium on Software Science and Engineering (SSE)*, RIMS, 1986, also Tech. Memorandum TM-0226, ICOT

[Ohsuga and Sakai 1988] Ohsuga, A. and Sakai, K., An efficient implementation method of reduction and narrowing in Metis, in *International Workshop of Unification (UNIF) '88*, 1988 also Tech. Report (to appear), ICOT

[Oka 1987] Oka, N., A Cognitive Model of Conscious/Unconscious Processing. in *4th Conf. Proc. Japan Soc. for Softw. Sc. Tech.*, pages 459–462, 1987 (in Japanese)

[Oka 1988] Oka, N., Cognitive Model of Conscious/Unconscious Processing and Its Simulation in a Parallel Logic Programming Language, Tech. Report TR-415, ICOT, 1988

[Okumura and Matsumoto 1987] Okumura, A. and Matsumoto, Y., Parallel Programming with Layered Streams, in *Proc. Fourth Symposium on Logic Programming*, San Francisco, 1987

[Prawitz 1965] Prawitz, D., *Natural Deduction*, Almquist and Wiksell, Stockholm, 1965

[Safra and Shapiro 1986] Safra, S. and Shapiro, E., Meta Interpreters for Real, in H.-J. Kugler, editor, *Information Processing 86*, pages 271–278, Dublin, Ireland, North-Holland, 1986

[Sakai 1988] Sakai, K., Toward Mechanization of Mathematics, in K. Fuchi and M. Nivat, editors, *Programming of Future Generation Computers*, pp.335–390, North-Holland, 1988

[Sato 1985] Sato, M., Typed Logical Calculus, Tech. Report 85-13, Department of Information Science, Faculty of Science, University of Tokyo, 1985

[Sato 1986] Sato, M., QJ: A Constructive Logical System with Types, in *France-Japan Artificial Intelligence and Computer Science Symposium 86*, Tokyo, 1986

[Sato 1987] Sato, M., Quty: A Concurrent Language Based on Logic and Function, in *Proc. Fourth International Conference on Logic Programming*, pp.1034–1056, MIT Press, 1987

[Satoh 1987] Satoh, K., Minimal change — A criterion for choosing between competing models —, Tech. Report TR-316, ICOT, 1987

[Satoh 1988] Satoh, K., Nonmonotonic reasoning by minimal belief revision, in this volume, 1988

[Sawamura and Minami 1988] Sawamura, H. and Minami, T., General-Purpose Reasoning Assistant System EUODHILOS and Its Applications, Tech. Memorandum TM-0576, ICOT, 1988

[Sawamura et al. 1988] Sawamura, H., Minami, T., Sato, K., and Tsuchiya, K., Potentials of General-Purpose Reasoning Assistant System EUODHILOS, in *Symposium on Software Science and Engineering (SSE)*, RIMS, 1988

[Shapiro 1982] Shapiro, E., *Algorithmic program debugging*, Ph.d thesis, Yale University Computer Science Dept., 1982, Published by MIT Press, 1983

[Smith 1984] Smith, B. C., Reflection and Semantics in Lisp, in *Proc. 11th Annual ACM Symp. on the Principles of Programming Languages*, pp.23–35, ACM, 1984

[Takayama 1987] Takayama, Y., Writing Programs as QJ-Proofs and Compiling into PROLOG Programs, in *Proc. 4th Symposium on Logic Programming*, San Francisco, 1987

[Takayama 1988a] Takayama, Y., QPC: QJ-based Proof Compiler —Simple Examples and Analysis, in *European Symposium on Programming '88*, Nancy, France, 1988

[Takayama 1988b] Takayama, Y., Proof Theoretic Approach to the Extraction of Redundancy-free Realizer Codes, (to appear), 1988

[Takeuchi and Furukawa 1986] Takeuchi, A. and Furukawa, K., Partial Evaluation of Prolog Programs and Its Application to Meta Programming, in H.-J. Kugler, editor, *Information Processing 86*, pages 415–420, Dublin, Ireland, North-Holland, 1986

[Takeuchi et al. 1987a] Takeuchi, A., Takahashi, K., and Shimizu, H., A Description Language with AND/OR Parallelism for Concurrent Systems and Its Stream-Based Realization, Tech. Report TR-229, ICOT, 1987

[Takeuchi et al. 1987b] Takeuchi, A., Takahashi, K., and Shimizu, H., A Parallel Problem Solving Language for Concurrent Systems, in *Proc. IFIP WG10.1*, 1987, (to appear)

[Tanaka 1988] Tanaka, J., Meta-Interpreters and Reflective Operations in GHC, in this volume, 1988

[Uchihira et al. 1987] Uchihira, N., Kasuya, T., Matsumoto, K., and Honiden, S., Concept Program Synthesis with Reusable Components Using Temporal Logic, Tech. Report TR-271, ICOT, 1987

[Ueda 1986a] Ueda, K., Introduction to Guarded Horn Clauses, Tech. Report TR-209, ICOT, 1986

[Ueda 1986b] Ueda, K., Guarded Horn Clauses: A Parallel Logic Programming Language with the Concept of a Guard, Tech. Report TR-208, ICOT, 1986 (revised 1987), also in K. Fuchi and M. Nivat, editors, *Programming of Future Generation Computers*, pp.441–456, North-Holland, 1988

[Ueda 1988] Ueda, K. and Furukawa, K., Transformation Rules for GHC Programs, in this volume, 1988

[Weyhrauch 1980] Weyhrauch, R. W., Prolegomena to a Theory of Mechanized Formal Reasoning, *Artif. Intell.*, 13(1–2):133–170, 1980